EECS 280 Final Exam
Fall 2019 SOLUTIONS

This is a closed-book exam. You may use one note sheet, 8.5"x11", double-sided, with your name on it. This exam has 5 problems on 18 pages, and it is out of 100 points total. **This is the exam booklet in which you will write your answers.**

Read the entire exam through before you begin working. Work on those problems you find easiest first. Read each question carefully, and note all that is required of you. Keep your answers clear and concise. Write your answers in the space provided. Assume all code is in standard C++11, and use only standard C++11 in your solutions. Throughout the exam, you may assume all necessary headers and the `using namespace std;` directive are present. You do not need to verify REQUIRES clauses with `assert` unless instructed to do so.

Write your uniqname on the line provided at the top of each page.

You are to abide by the University of Michigan/Engineering honor code. To receive a grade, sign below to signify that you have kept the honor code pledge:

I have neither given nor received aid on this exam, nor have I concealed any violations of the Honor Code.

Signature: _____

Name: _____

Uniqname: _____

UMID: _____

Exam Room Number: _____

First and last name of person sitting to your **right** (write the symbol ⊥ if at the end of a row):

_____

First and last name of person sitting to your **left** (write the symbol ⊥ if at the end of a row):

_____

Correctly completing this cover page is worth **1 point**.

Problem 1: Short Answers [22 points]

1a) [8 points] True/False. Mark the bubble corresponding with your answer under each question. If you change your answer, be sure to completely erase the bubble.

i) Any recursive solution written for a binary search tree will always be tree recursive.

FALSE

ii) When the `new` keyword is used to create a dynamic object, the `new` operation evaluates to the value of that object.

FALSE

iii) In C++, static member functions have access to the `this` pointer.

FALSE

iv) The private members of a class can never be accessed by any other class.

FALSE

v) A container of pointers is responsible for deleting memory associated with those pointers.

FALSE

vi) Imposter syndrome is a condition in which someone believes that they know less than other people in a similar situation.

TRUE

vii) A function that is linear recursive can also be tail recursive.

TRUE

viii) An uncaught exception will cause the program to terminate.

TRUE

1b) [6 points]

 See the **Section 1 Reference Sheet** in the Reference material. 

Consider the code segment provided in the Reference Sheet Section 1b) and fill in the box next to the **`main`** function below with what will be printed when that code is run. Write "nothing" if nothing is printed. Ignore any memory leaks. Whitespace will not be graded.

`int main() {`
`  Apple *fuji = new Apple();`
`  Apple *golden = fuji`
`  Fruit *eatme = fuji;`
`  delete eatme;`
`}`
	fruit ctor
	apple ctor
	apple dtor
	fruit dtor

`int main() {`
`  Apple gala;`
`  Fruit anyfruit = gala;`
`  Apple goldspur = gala;`
`}`
	fruit ctor
	apple ctor
	
	fruit copy ctor
	
	fruit ctor
	apple copy ctor
	
	apple dtor
	fruit dtor
	
	fruit dtor
	
	apple dtor
	fruit dtor

1c) [8 points]

For each of the following inputs to standard input, which of the following is the output to standard output?

ⓐ ⓑ ⓒ ⓓ

1. `5 Tangier Tangier`
   a. `We can't book your trip.`
   b. `Source and Destination are identical`
      `Error encountered.`
   c. `Source and Destination are identical`
      `Invalid input.`
   d. `Invalid input.`

ⓐ ⓑ ⓒ ⓓ

2. `-2 Chengdu Sacramento`
   a. `We can't book your trip.`
      `After makeTrip()`
   b. `Invalid input.`
   c. `We can’t book your trip.`
   d. `We can’t book your trip.`
      `Invalid input.`

ⓐ ⓑ ⓒ ⓓ

3. `67 Seattle Seattle`
   a. `We can't book your trip.`
      `Source and Destination are identical`
   b. `We can’t book your trip.`
      `Source and Destination are identical`
      `Error encountered.`
   c. `We can’t book your trip.`
   d. `We can’t book your trip.`
      `Source and Destination are identical`
      `Invalid input.`

ⓐ ⓑ ⓒ ⓓ

4. `2 Kiev Reykjavik`
   a. `Booking finished!`
   b. `Booking finished!`
      `After makeTrip()`
   c. `We can’t book your trip.`
   d. `Booking finished!`

```
Invalid input.
```

Problem 2: Dynamic Memory (14 points)

 See the **Section 2 Reference Sheet** in the Reference material. 

2a) [6 points] Implement the custom constructor for the Directory class. The constructor must initialize the new Directory's name to be equal to *name_in* and the number of files in the Directory to *files_in*. The Directory has an initial capacity of 100. The constructor takes in a vector of files and constructs a Directory object containing those files.

Your solution must use an initializer list where possible.

Directory::Directory (const string &name_in, const vector<File> &files_in):

```
 : dir_name(name_in), num_files(files_in.size()), capacity(100)
```

{

```
file_arr = new File [capacity];
for (int i = 0; i < files_in.size(); ++i)
        file_arr[i] = files_in[i];
```

}

2b) [3 points] The following implementation of the shrink function in the Directory class was implemented **incorrectly** and contain errors. Fill in the box(es) next to the corresponding error(s) present in this implementation.

```
void Directory::shrink() {
      capacity = num_files;
      File* new_files = new File();
      for(int i = 0; i < num_files; i++) {
            new_files[i] = File( file_arr[i] );
      }
      file_arr = new_files;
      dir_name += “_compressed”;
}
```

☐	Local variables not copied
☐	Existing dynamic memory not deleted
☐	Dynamic memory not correctly copied
☐	Dynamic memory not correctly allocated

2c) [3 points] The following implementation of the overloaded Assignment operator in the Directory class was implemented **incorrectly** and contains errors. Fill in the box(es) next to the corresponding error(s) present in this implementation.

```
Directory& Directory::operator= (const Directory & other){
     if (this == &other) {
        return *this;
     }
     delete[] file_arr;
     file_arr = new File[capacity];
     for (int i = 0; i < capacity; ++i) {
        file_arr[i] = other.file_arr[i];

     }
     return *this;
}
```

☐	Local variables not copied
☐	Existing dynamic memory incorrectly deleted
☐	Does not properly implement deep copy of dynamic memory
☐	Invalid check for self-assignment

2d) [2 points] Implement the destructor for the Directory class. This destructor must ensure no memory is leaked when it runs.

```
Directory::~Directory() {
    delete[] file_arr;
}
```

Problem 3: Lists and Templates (24 points)

 See the **Section 3 Reference Sheet** in the Reference material. 

3a) [10 points] Implement the **`first_out_of_order`** function below. This function finds the index of the first element out of order in a doubly linked list. Out of order is defined as an element that has a value less than the element before it (a sorted list would be sorted least to greatest). Your solution may **NOT** use any standard library features. Iterators are **not** allowed. Your solution may **NOT** use recursion. You **MUST** use **ONLY** the greater-than operator (>) for comparisons. Other comparison operators will not be allowed.

```
// REQUIRES: Type T has implemented the greater-than(>) comparison operator.
// EFFECTS:  Returns the index of the first element out of order  (lesser
value than the element before it). If the list is already sorted, return -1.
// EXAMPLES:
// {1, 2, 3, 6, 5, 8} => 4
// {2, 3, 6, 4, 9, 8} => 3
// {4, 4, 3, 9, 2, 0, 0} => 2
```

```
int List::first_out_of_order() {
    if (first == nullptr || first == last) {
            return -1;
     }
     Node *n = first;
     int index = 1;
     while (n->next != nullptr) {
         if (n->datum > n->next->datum) {
             return index;
         }
         n = n->next;
         ++index;
     }
     return -1;
}
```

3b) [8 points] Implement the following **`remove_index`** function as described by its RME. You may **NOT** use standard library features or recursion in your solution. Iterators are **not** allowed. You may **NOT** use any functions except those defined in the Reference Sheet. You may assume the `pop_back` and `pop_front` functions are implemented.

```
// REQUIRES: index is a valid index in the list.
//           The list is not empty.
// EFFECTS:  Removes the Node at the given index while maintaining the
list properties.
// EXAMPLES:
// Given {1, 3, 5, 1, 8}, remove_index(3) => {1, 3, 5, 8}
// Given {1, 2, 3}, remove_index(0) => {2, 3}
// Given {1, 3}, remove_index(1) => {1}

void List::remove_index(int index) {
```

```
        Node *victim = first;

        if (index == 0) {
            pop_front();
            return;
        }

        for (int i = 0; i < index; ++i) {
            victim = victim->next;
        }

        if (victim == last) {
            pop_back();
            return;
        }

        victim->next->prev = victim->prev;
        victim->prev->next = victim->next;
        delete victim;
```

```
}
```

3c) [6 points] Implement the **`remove_all_out_of_order`** function according to its RME. You may **NOT** use recursion in your answer. You **MUST** use **`first_out_of_order`** and **`remove_index`** from **3a)** and **3b)** in your solution. You may assume that they are implemented correctly.

```
// EFFECTS: Removes all Nodes in the list whose value is out-of-order
//          (as defined in 3a).
// EXAMPLES:
// {1, 2, 4, 3, 5} => {1, 2, 4, 5}
// {1, 1, 7} => {1, 1, 7}
// {1, 3, 2, 1, 5, 1, 7, 6} => {1, 3, 5, 7}
```

```
void List::remove_all_out_of_order() {
```

```
        int index = first_out_of_order();
        while (index != -1) {
            remove_index(index);
            index = first_out_of_order();
        }
```

```
}
```

Problem 4: Iterators and Functors (20 points)

4a) [6 points] Declare and implement the ***function call operator*** for the functor class SquareSum as described by its RME. You may **NOT** use standard library features or recursion in your solution.

```
class SquareSum {
      int sum = 0;
public:
      // REQUIRES: Accepts one int N as its argument.
      // MODIFIES: sum
      // EFFECTS:  Adds the square of N to sum, then returns the new
      //           value of sum.
      // EXAMPLE:  sum = 3, N = 2 => 7 // sum = 3 + (2 * 2) = 7
```

```
int operator()(int N) {
        sum += N * N;
        return sum;
    }
```

```
};
```

4b) [7 points] Implement the **`traverse`** function below according to its RME. Your solution may **NOT** use any standard library features. Your solution may **NOT** use recursion.

```
// REQUIRES: Iterator supports the *, (prefix) ++, ==, and !=
//           operators.
//           The provided iterators point to elements of type int.
//           func takes one argument of type int, and returns an int.
//           begin != end
// EFFECTS:
//           Applies func to every element in the range [begin, end).
//           Returns the last value returned by a call to func.
//           Note: The elements in the range are not modified by func.
// EXAMPLES:
//           Note: element ranges represented with braced lists: {...}
//           traverse({1, 3}, fn) => fn(3) // calls fn(1), fn(3);
//           traverse({10}, fn) => fn(10) // calls fn(10);

template<typename Iterator, typename Functor>
int traverse(Iterator begin, Iterator end, Functor func) {
```

```
int result = *begin;
while (begin != end) {
    result = func(*begin);
    ++ begin;
}
return result;
```

```
}
```

4c) [7 points] Implement the **`magnitude`** function according to its RME. You may **NOT** use recursion in your answer. You **MUST** use traverse and SquareSum in your solution. You **MUST** use the **`sqrt`** function from the standard library to help you compute the square root.

```
// REQUIRES: vec contains at least one element.
// EFFECTS:  Computes the magnitude of vec. The magnitude of a vector
//           is defined as the square root of the sum of squares of
//           its elements.
//           magnitude(X1, X2, ... Xn) = sqrt(X1*X1 + X2*X2 + ... +
//                                                            Xn*Xn)
// EXAMPLES:
//           magnitude({29}) => 29     // sqrt(29*29)
//           magnitude({3, 4}) => 5    // sqrt(3*3 + 4*4)
//           magnitude({1, 2, 2}) => 3 // sqrt(1*1 + 2*2 + 2*2)

double magnitude(const vector<int> &vec) {
```

```
    SquareSum func;
    int sum = traverse(vec.begin(), vec.end(), func);
    return sqrt(sum);
```

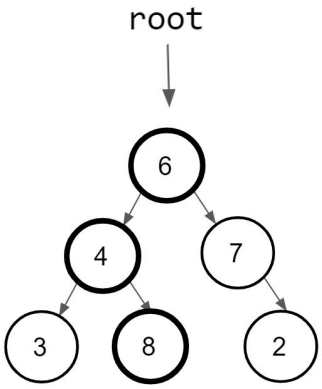
```
}
```

Problem 5: Recursion (19 Points)

For the following two questions you will work directly with the Node struct to the right, representing a tree containing ints. This tree is not sorted, has no duplicates and contains only positive ints. You may **NOT** use any functions except those defined in this problem. Your functions **MUST** be **recursive** and you may **NOT** use loops.

```
struct Node{
  int datum;
  Node * right;
  Node * left;
};
```

5a) [8 points] Implement the **greatest_path** function below according to its RME. You may make use of the max function provided below.



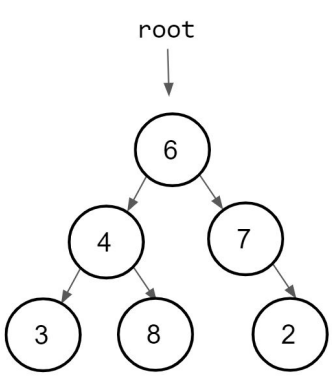

```
//EFFECTS: returns a if a > b, returns b if
//         b > a
     int max(int a, int b);

//EFFECTS: Returns the greatest sum of the
//         path starting at the root of the
//         tree and ending at a leaf node. A
//         leaf node has a datum but no left
//         or right child.
//         Returns 0 for an empty tree.
//EXAMPLE: For the tree on the right, greatest_path
//         returns (6 + 4 + 8) = 18

int greatest_path(const Node *root){
```

```
    if(!root)   {
        return 0;
    }
    int left_sum = greatest_path(root->left) +
        root->datum;
    int right_sum = greatest_path(root->right) +
        root->datum;
    return max(left_sum, right_sum);
```

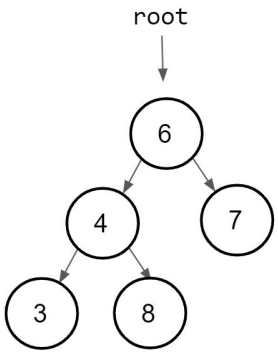
```
}
```

5b) [7 points] Implement the **check_tree_nodes** function below according to its RME.

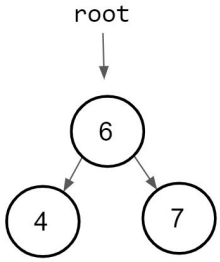
```
// EFFECTS:  Returns true if every node has exactly two children or is
//           a leaf node.
// EXAMPLES:
```



False



True



True

True

```
bool check_tree_nodes(const Node * root){
      if(!root){
            return true;
      }
      if(!root->left && root->right){
            return false;
      }
      if(!root->right && root->left){
            return false;
      }
      return check_tree_nodes(root->right) &&
             check_tree_nodes(root->left);

}
```

5c) [4 points] For each of the functions below, select the property that best describes the Function. Fill in only **one** bubble for each function. Make sure to erase completely if you change your answer.

i)
```
int combination(int n, int r) {
    if (r == 0 || n == r) {
        return 1;
    }
    else {
        return combination(n - 1, r) + combination(n - 1, r - 1);
    }
}
```

Tail Recursive	Tree Recursive	Recursive (Not Tail/Tree)	Not Recursive
○	○	○	○

Tree Recursive

ii)
```
int potential_function(int x, int y)    {
    if (x == 0 || y == 0) {
        return 0;
    }
    if (x == y)     {
        return potential_function(x - 1, y + 1);
    }
    if (x > y)      {
        return potential_function(x - 1, y);
    }
    return potential_function(x, y - 1);
}
```

Tail Recursive	Tree Recursive	Recursive (Not Tail/Tree)	Not Recursive
○	○	○	○

Tail Recursive